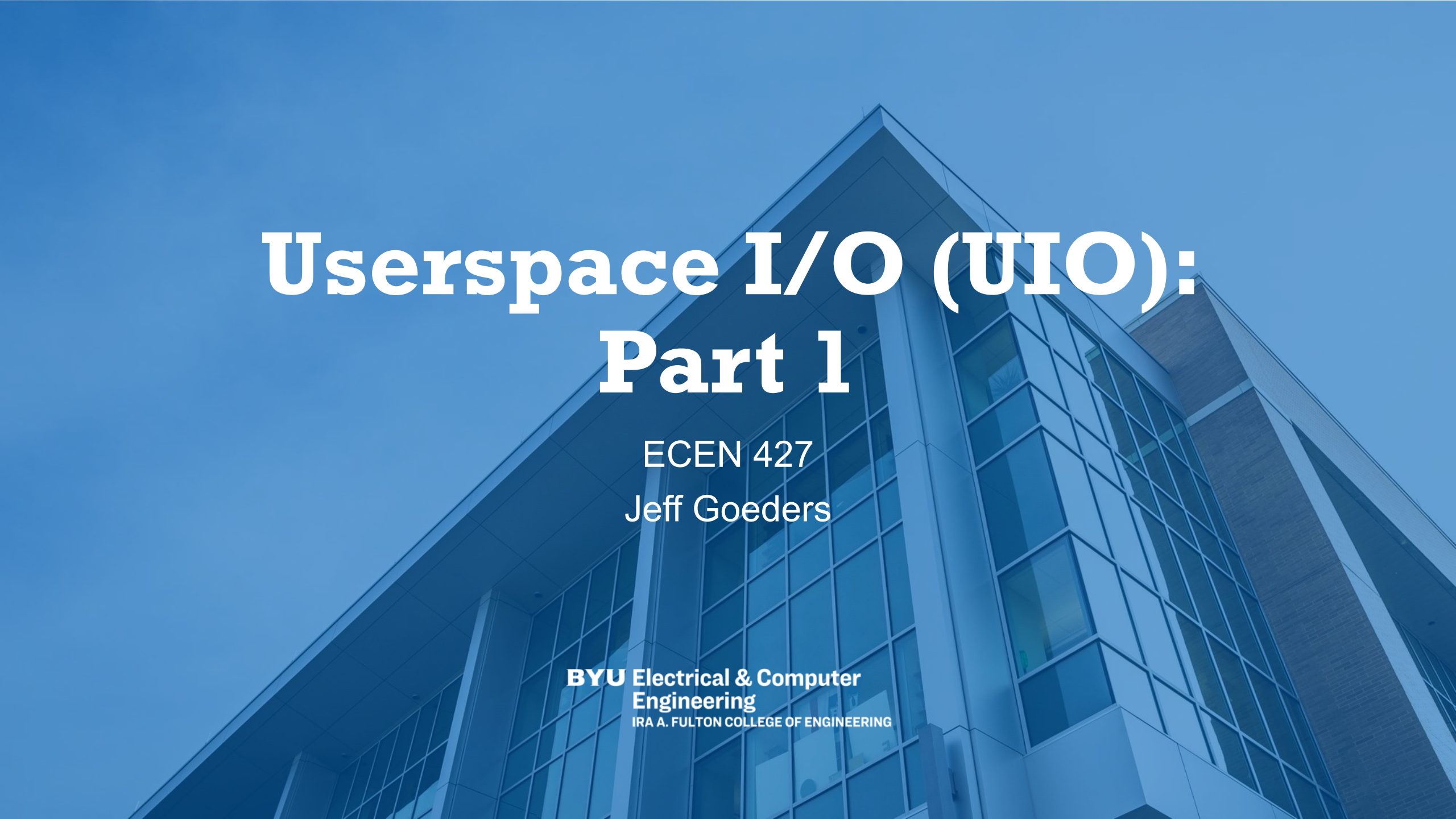


Userspace I/O (UIO): Part 1

ECEN 427

Jeff Goeders

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

Learning Objectives

By the end of this lecture, you should be able to:

- Explain the challenges of accessing hardware devices from user space
- Describe how Linux exposes hardware devices to user space programs
- Write user space code that reads and writes device registers
- Safely use pointers to access device registers

What is a device driver?

- Software that knows how to **talk to a specific hardware device**
  - Knows the device's registers, what to write where, and what the values mean
- Provides an **abstraction;** exposes a clean interface and hides the messy hardware details
  - The rest of the system doesn't need to know *how* the UART works, just how to send/receive bytes
  - This abstraction is an API (application programming interface)
  - Defined by the header file
- Typically runs in the **kernel**, since it needs privileged access to hardware
  - ...but not always — that's today's topic

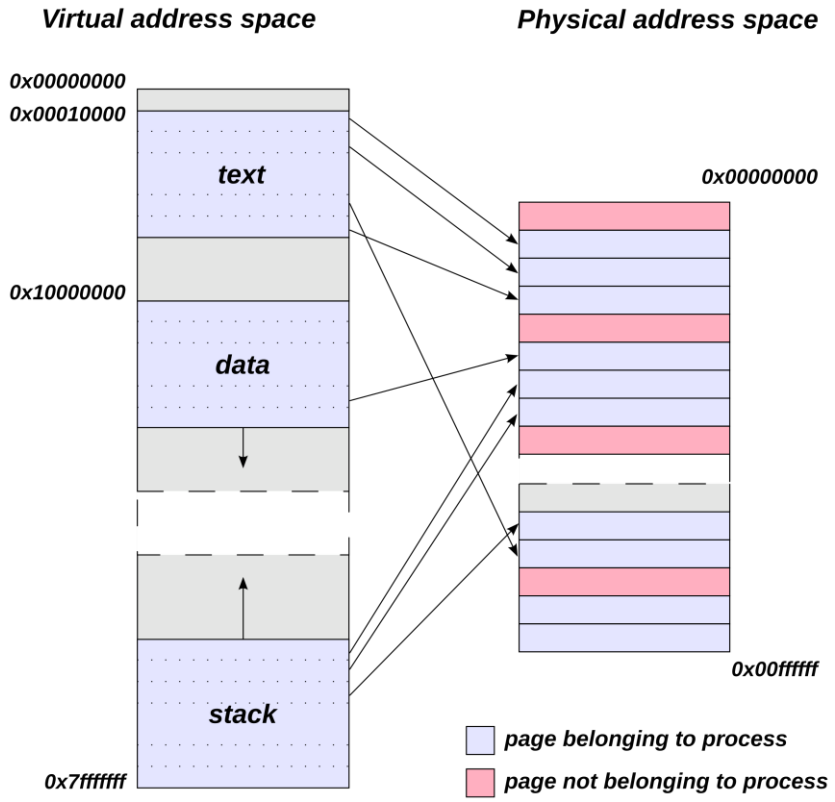
AXI GPIO Device Registers

Offset	Register	Meaning
0x000	GPIO_DATA	Pin values
0x004	GPIO_TRI	Direction (not present on our blocks)
0x11C	GIER	Global interrupt enable
0x120	IP_ISR	Interrupt status
0x128	IP_IER	Interrupt enable

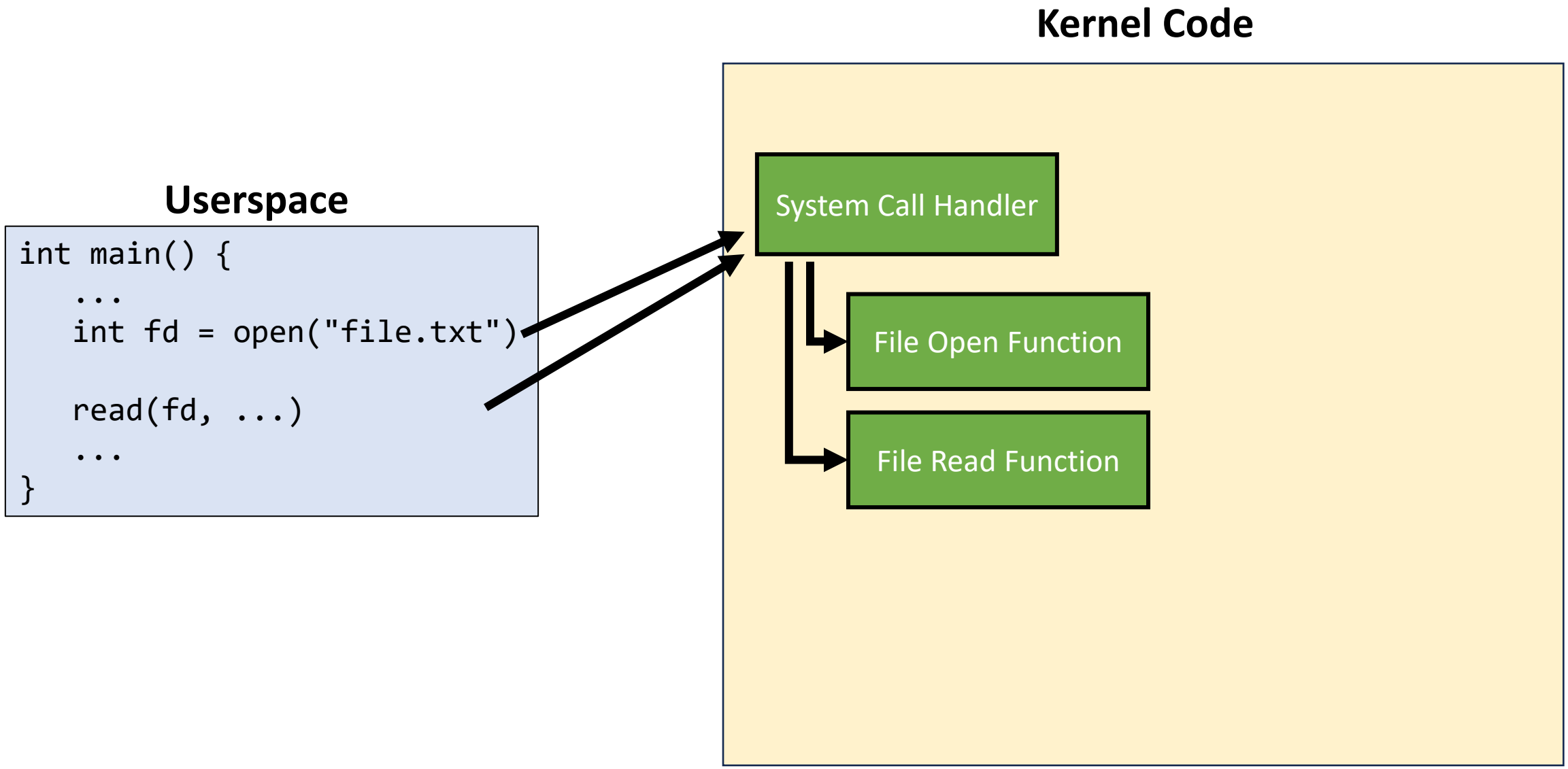
Devices & User Space

- **Recall**: From user space, a process can only talk to physical addresses mapped to a virtual address
- **Problem:** By default, device addresses are not mapped to the user virtual address space.
- **Solution:** We need the help of code within the Kernel to interface with hardware (i.e. kernel drivers)

Remember, system calls are how we communicate with the kernel from user space.



Linux System Calls



System Calls

At the end of the OS lecture we looked at the list of syscalls:

- https://chromium.googlesource.com/chromiumos/docs/+/HEAD/constants/syscalls.md

Where are the system calls to talk to hardware devices?

- UART
- Disk
- Video card
- Physical memory
- USB devices

Invoking Driver Code From User Space?

User space

```
int main() {
    ...
    uart_get_input() ???
    ...
}
```

Don't kernel device drivers provide functions that we can call to access the devices?

No! This would require adding new system calls with every new driver in the kernel.

There are currently **thousands** of device drivers in the Linux kernel.

Device Files

- Look like ordinary files in the filesystem, but are special files
- They are **interfaces to the driver** that controls the device
- By using **system calls that operate on files**, our code can interface with the driver.
- They are not real files…

Have you used a device file before?

You used /dev/ttyUSB1 to talk to the serial driver during Lab 1.

...an aside...pseudo-devices

Pseudo-devices [edit]

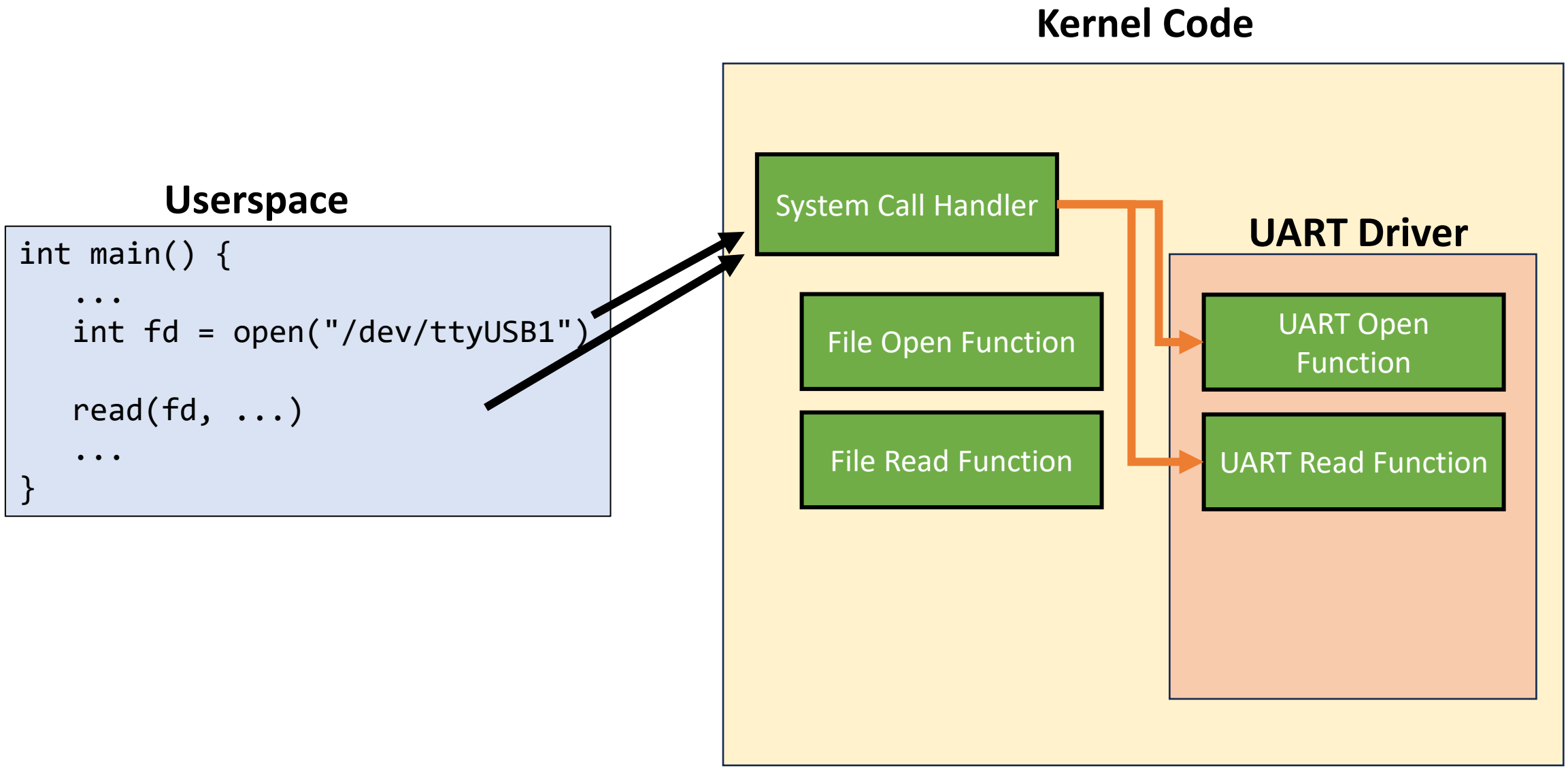
Device nodes on Unix-like systems do not necessarily have to correspond to physical devices. Nodes that lack this correspondence form the group of *pseudo-devices*. They provide various functions handled by the operating system. Some of the most commonly used (character-based) pseudo-devices include:

- `/dev/null` – accepts and discards all input written to it; provides an end-of-file indication when read from.
- `/dev/zero` – accepts and discards all input written to it; produces a continuous stream of null characters (zero-value bytes) as output when read from.
- `/dev/full` – produces a continuous stream of null characters (zero-value bytes) as output when read from, and generates an `ENOSPC` ("disk full") error when attempting to write to it.
- `/dev/random` – produces bytes generated by the kernel's cryptographically secure pseudorandom number generator. Its exact behavior varies by implementation, and sometimes variants such as `/dev/urandom` or `/dev/arandom` are also provided.
- `/dev/stdin`, `/dev/stdout`, `/dev/stderr` – access the process's standard streams.
- `/dev/fd/`*n* – accesses the process's file descriptor *n*.

Additionally, BSD-specific pseudo-devices with an `ioctl` interface may also include:

- `/dev/pf` – allows userland processes to control PF through an `ioctl` interface.
- `/dev/bio` – provides `ioctl` access to devices otherwise not found as `/dev` nodes, used by `bioctl` to implement RAID management in OpenBSD and NetBSD.
- `/dev/sysmon` – used by NetBSD's envsys framework for hardware monitoring, accessed in the userland through `proplib(3)` by the `envstat` utility.[8]

Linux System Calls



A Driver for Every Device?

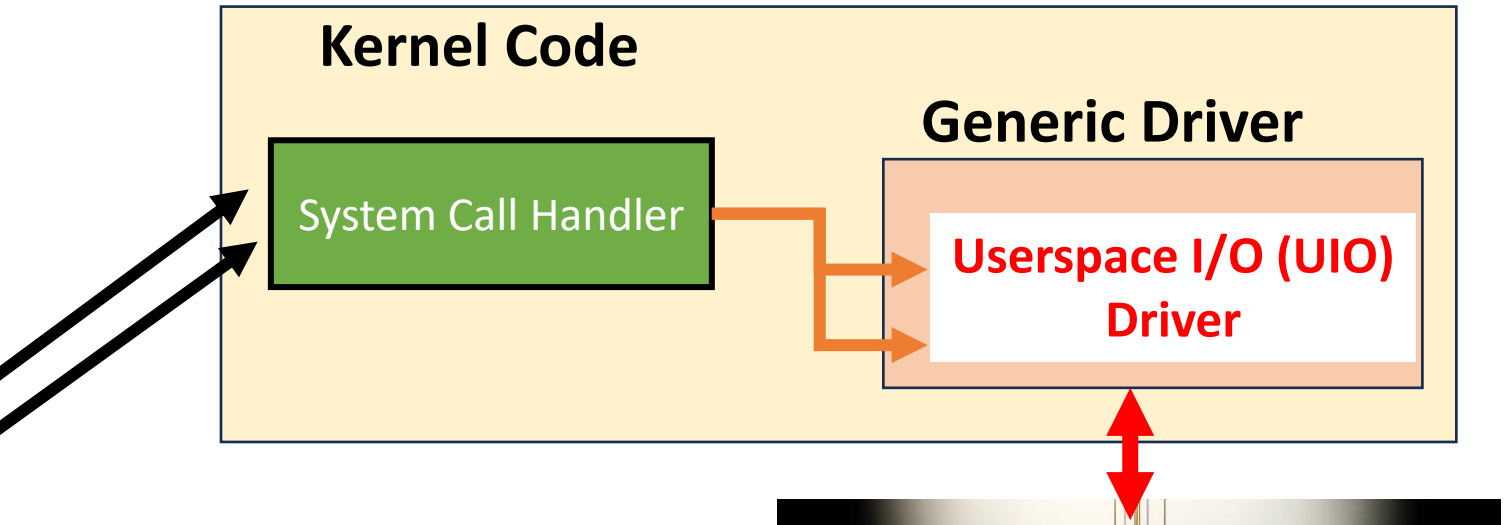
Do we really need a separate kernel driver for every hardware device?

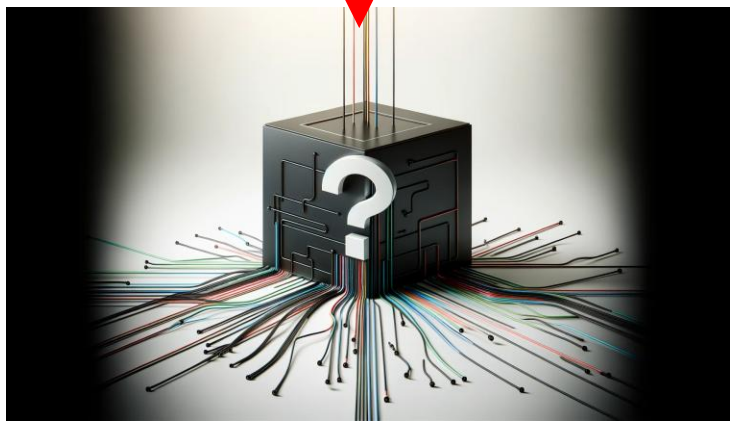
Can we make a generic driver that would **allow user space to talk to different kinds of hardware**?

User space

```
int main() {
    ...
    int fd = open("/dev/...")

    ?????(..., ...)
    ...
}
```





What system calls would you use to interface with this driver?

A new syscall approaches…

It can be confusing that the UIO driver doesn't use the **`read()`** and **`write()`** syscalls to trigger reads and writes to device registers.

It will make more sense once we learn about a new syscall…

`mmap()`

Map files (or devices) into memory

Example…

Userspace I/O (UIO) Driver

- The Linux **Userspace I/O (UIO) driver**, provides generic access to hardware devices allowing user space to:
  - Access device memory addresses (read/write regs)
  - Check if device generated an interrupt (***coming later…***)

- Devices managed by the UIO driver will have a device file:

  `/dev/uioX`

- UIO is not meant to serve as the device driver. It's an interface to allow for user space drivers.

Userspace I/O (UIO) Driver Example

We can open() the file, then mmap() using the file descriptor to gain a pointer that can be used to access device registers:

```
int fd = open("/dev/uio0", O_RDWR);
char *p = mmap(NULL, 0x1000, PROT_READ | PROT_WRITE,
               MAP_SHARED, fd, 0);
printf("GPIO_DATA value: %u\n", *(volatile uint32_t*)p);
```

Offset	Register	Meaning
0x000	GPIO_DATA	Pin values
0x004	GPIO_TRI	Direction (not present on our blocks)
0x11C	GIER	Global interrupt enable
0x120	IP_ISR	Interrupt status
0x128	IP_IER	Interrupt enable

Why the odd pointer casting?

- Recall, use volatile to ensure we force a read on the memory bus

…let's talk about the pointer size

Caution: Pointer Arithmetic

What does this print?

```
uint32_t * p = (uint32_t*) 0x1000;
p = p + 1;
printf("p points to 0x%x\n", p);
```

It prints: 0x1004

Why?

- Pointer arithmetic is designed to work with arrays of the data type.
- Address increments by the size of the data type.

Be careful:

- Device documents list register offsets in bytes (absolute address)
- Registers are 32-bits, so it makes sense to use a 32-bit pointer type (e.g., uint32_t*)
- If you add the offset to a uint32_t *, you will have a bad time!!

- Consider using one of these approaches:
  - Divide offsets by 4
  - Use uint8_t* and cast to uint32_t* before use

Lab 2

In lab 2, you will create user space drivers (interfacing with UIO) for:

- buttons (milestone 1)
- switches (milestone 1)
- LEDs (milestone 1)
- interrupt controller (milestone 2)

Example code for interacting with UIO:

https://github.com/byu-cpe/ecen427_student/blob/main/userspace/drivers/uio_example/generic_uio_example.c